Chapter 11

The Unavoidable Math Chapter

In This Chapter

- Using the ++ and –– operators
- Making the modulus useful
- Employing various operator shortcuts
- Working with various math functions
- Creating random numbers
- Understanding the order of precedence

One of the reasons I shunned computers in my early life was that I feared the math. Eventually, I learned that math doesn't play a big role in programming. On one hand, you need to know *some* math, especially when a program involves complex calculations. On the other hand, it's the computer that does the math — you just punch in the formula.

In the programming universe, math is necessary but painless. Most programs involve some form of simple math. Graphics programming uses a lot of math. And games wouldn't be interesting if it weren't for random numbers. All that stuff is math. I believe that you'll find it more interesting than dreadful.

Math Operators from Beyond Infinity

Two things make math happen in C programming. The first are the math operators, which allow you to construct mathematical equations and formulas. These are shown in Table 11-1. The second are math functions, which implement complex calculations by using a single word. To list all those functions in a table would occupy a lot of space.

Table 11-1 C Math Operators

Operator	*Function*	*Example*
+	Addition	var=a+b
–	Subtraction	var=a-b
*	Multiplication	var=a*b
/	Division	var=a/b
%	Modulo	var=a%b
++	Increment	var++
--	Decrement	var--
+	Unary plus	+var
–	Unary minus	-var

- Chapter 5 introduces the basic math operators: +, –, *, and /. The rest aren't too heavy-duty to understand — even the malevolent modulo.
- The C language comparison operators are used for making decisions. Refer to Chapter 8 for a list.
- Logical operators are also covered in Chapter 8.
- The single equal sign, =, is an operator, although not a mathematical operator. It's the *assignment* operator, used to stuff a value into a variable.
- Bitwise operators, which manipulate individual bits in a value, are covered in Chapter 17.
- Appendix C lists all the C language operators.

Incrementing and decrementing

Here's a handy trick, especially for those loops in your code: the increment and decrement operators. They're insanely useful.

To add one to a variable's value, use ++, as in:

```
var++;
```

After this statement is executed, the value of variable *var* is increased (incremented) by 1. It's the same as writing this code:

```
var=var+1;
```

You'll find ++ used all over, especially in `for` loops; for example:

```
for(x=0;x<100;x++)
```

This looping statement repeats 100 times. It's much cleaner than writing the alternative:

```
for(x=0;x<100;x=x+1)
```

Exercise 11-1: Code a program that displays this phrase ten times: "Get off my lawn, you kids!" Use the incrementing operator `++` in the looping statement.

Exercise 11-2: Recode your answer for Exercise 11-1 using a `while` loop if you used a `for` loop, or vice versa.

The `++` operator's opposite is the decrementing operator `--`, which is two minus signs. This operator decreases the value of a variable by 1; for example:

```
var--;
```

The preceding statement is the same as

```
var=var-1;
```

Exercise 11-3: Write a program that displays values from `-5` through `5` and then back to `-5` in increments of 1. The output should look like this:

```
-5 -4 -3 -2 -1 0 1 2 3 4 5 4 3 2 1 0 -1 -2 -3 -4 -5
```

This program can be a bit tricky, so rather than have you look up my solution on the web, I'm illustrating it in Listing 11-1. Please don't look ahead until you've attempted to solve Exercise 11-3 on your own.

Listing 11-1: Counting Up and Down

```
#include <stdio.h>

int main()
{
    int c;

    for(c=-5;c<5;c++)
        printf("%d ",c);
    for(;c>=-5;c--)
        printf("%d ",c);
    putchar('\n');
    return(0);
}
```

The crux of what I want you to see happens at Line 9 in Listing 11-1, but it also plays heavily off the first `for` statement at Line 7. You might suspect

that a loop counting from -5 to 5 would have the value 5 as its stop condition, as in:

```
for(c=-5;c<=5;c++)
```

The problem with this construct is that the value of c is incremented to trigger the end of the loop, which means that c equals 6 when the loop is done. If c remains less than 5, as is done at Line 7, then c is automatically set to 5 when the second loop starts. Therefore, in Line 9, no initialization of the variable in the for statement is necessary.

Exercise 11-4: Construct a program that displays values from -10 to 10 and then back down to -10. Step in increments of 1, as was done in Listing 11-1, but use two while loops to display the values.

Prefixing the ++ and -- operators

The ++ operator always increments a variable's value, and the -- operator always decrements. Knowing that, consider this statement:

```
a=b++;
```

If the value of variable b is 16, you know that its value will be 17 after the ++ operation. So what's the value of variable a — 16 or 17?

Generally speaking, C language math equations are read from left to right. (Refer to the later section "The Holy Order of Precedence" for specifics.) Based on this rule, after the preceding statement executes, the value of variable a is 16, and the value of variable b is 17. Right?

The source code in Listing 11-2 helps answer the question of what happens to variable a when you increment variable b on the right side of the equal sign (the assignment operator).

Listing 11-2: What Comes First — the = or the ++?

```
#include <stdio.h>

int main()
{
    int a,b;

    b=16;
    printf("Before, a is unassigned and b=%d\n",b);
    a=b++;
    printf("After, a=%d and b=%d\n",a,b);
    return(0);
}
```

Exercise 11-5: Type the source code from Listing 11-2 into a new project. Build and run.

When you place the ++ or -- operator after a variable, it's called *post-incrementing* or *post-decrementing,* respectively. If you want to increment or decrement the variable before it's used, you place ++ or -- *before* the variable name; for example:

```
a=++b;
```

In the preceding line, the value of `b` is incremented, and then it's assigned to variable `a`. Exercise 11-6 demonstrates.

Exercise 11-6: Rewrite the source code from Listing 11-2 so that the equation in Line 9 increments the value of variable `b` before it's assigned to variable `a`.

And what of this monster:

```
a=++b++;
```

Never mind! The ++*var*++ thing is an error.

Discovering the remainder (modulus)

Of all the basic math operator symbols, % is most likely the strangest. No, it's not the percentage operator. It's the *modulus* operator. It calculates the remainder of one number divided by another, which is something easier to show than to discuss.

Listing 11-3 codes a program that lists the results of modulus 5 and a bunch of other values, ranging from `0` through `29`. The value `5` is a constant, defined in Line 3 in the program. That way, you can easily change it later.

Listing 11-3: Displaying Modulus Values

```
#include <stdio.h>

#define VALUE 5

int main()
{
    int a;

    printf("Modulus %d:\n",VALUE);
    for(a=0;a<30;a++)
        printf("%d %% %d = %d\n",a,VALUE,a%VALUE);
    return(0);
}
```

Line 11 displays the modulus results. The `%%` placeholder merely displays the `%` character, so don't let it throw you.

Exercise 11-7: Type the source code from Listing 11-3 into a new project. Build and run.

Now that you can see the output, I can better explain that a modulus operation displays the remainder of the first value divided by the second. So `20 % 5` is `0`, but `21 % 5` is `1`.

Exercise 11-8: Change the `VALUE` constant in Listing 11-3 to 3. Build and run.

Saving time with assignment operators

If you're a fan of the ++ and -- operators (and I certainly am), you'll enjoy the operators listed in Table 11-2. They're the math assignment operators, and like the increment and decrement operators, not only do they do something useful, but they also look really cool and confusing in your code.

Table 11-2 C Math Assignment Operators

Operator	*Function*	*Shortcut for*	*Example*
+=	Addition	x=x+n	x+=n
-=	Subtraction	x=x-n	x-=n
*=	Multiplication	x=x*n	x*=n
/=	Division	x=x/n	x/=n
%=	Modulo	x=x%n	x%=n

Math assignment operators do nothing new, but they work in a special way. Quite often in C, you need to modify a variable's value. For example:

```
alpha=alpha+10;
```

This statement increases the value of variable *alpha* by 10. In C, you can write the same statement by using an assignment operator as follows:

```
alpha+=10;
```

Both versions of this statement accomplish the same thing, but the second example is more punchy and cryptic, which seems to delight most C programmers. See Listing 11-4.

Listing 11-4: Assignment Operator Heaven

```
#include <stdio.h>

int main()
{
    float alpha;

    alpha=501;
    printf("alpha = %.1f\n",alpha);
    alpha=alpha+99;
    printf("alpha = %.1f\n",alpha);
    alpha=alpha-250;
    printf("alpha = %.1f\n",alpha);
    alpha=alpha/82;
    printf("alpha = %.1f\n",alpha);
    alpha=alpha*4.3;
    printf("alpha = %.1f\n",alpha);
    return(0);
}
```

Exercise 11-9: Type the source code from Listing 11-4 into your text editor. Change Lines 9, 11, 13, and 15 so that assignment operators are used. Build and run.

When you use the assignment operator, keep in mind that the = character comes *last*. You can easily remember this tip by swapping the operators; for example:

```
alpha=-10;
```

This statement assigns the value `-10` to the variable *`alpha`*. But the statement

```
alpha-=10;
```

decreases the value of *`alpha`* by 10.

Exercise 11-10: Write a program that outputs the numbers from 5 through 100 in increments of 5.

Math Function Mania

Beyond operators, math in the C language is done by employing various mathematical functions. So when you're desperate to find the arctangent of a value, you can whip out the `atan()` function and, well, there you go.

Most math functions require including the `math.h` header file in your code. Some functions may also require the `stdlib.h` header file, where stdlib means *st*andard *lib*rary.

Exploring some common math functions

Not everyone is going to employ their C language programming skills to help pilot a rocket safely across space and into orbit around Titan. No, it's more likely that you'll attempt something far more down-to-earth. Either way, the work will most likely be done by employing math functions. I've listed some common ones in Table 11-3.

Table 11-3 Common, Sane Math Functions

Function	*#include*	*What It Does*
sqrt()	math.h	Calculates the square root of a floating-point value
pow()	math.h	Returns the result of a floating-point value raised to a certain power
abs()	stdlib.h	Returns the absolute value (positive value) of an integer
floor()	math.h	Rounds up a floating-point value to the next whole number (nonfractional) value
ceil()	math.h	Rounds down a floating-point value to the next whole number

All the functions listed in Table 11-2, save for the `abs()` function, deal with floating-point values. The `abs()` function works only with integers.

You can look up function references in the man pages, accessed via Code::Blocks or found online or at the command prompt in a Unix terminal window.

Listing 11-5 is littered with a smattering of math functions from Table 11-3. The compiler enjoys seeing these functions, as long as you remember to include the `math.h` header file at Line 2.

Listing 11-5: Math Mania Mangled

```
#include <stdio.h>
#include <math.h>

int main()
{
    float result,value;

    printf("Input a float value: ");
    scanf("%f",&value);
    result = sqrt(value);
    printf("The square root of %.2f is %.2f\n",
           value,result);
    result = pow(value,3);
    printf("%.2f to the 3rd power is %.2f\n",
           value,result);
    result = floor(value);
    printf("The floor of %.2f is %.2f\n",
           value,result);
    result = ceil(value);
    printf("And the ceiling of %.2f is %.2f\n",
           value,result);
    return(0);
}
```

Exercise 11-11: Create a new project using the source code from Listing 11-5. Be aware that I've wrapped the `printf()` functions in the listing so that they're split between two lines; you don't need to wrap them in your source code. Build the project. Run it and try various values as input to peruse the results.

Exercise 11-12: Write a program that displays the powers of 2, showing all values from 2^0 through 2^{10}. These are the Holy Numbers of Computing.

- The math functions listing in Table 11-3 are only a small sampling of the variety available.
- Generally speaking, if your code requires some sort of mathematical operation, check the C library documentation, the man pages, to see whether that specific function exists.
- On a Unix system, type **man 3 math** to see a list of the C library's math functions.

- The `ceil()` function is pronounced "seal." It's from the word *ceiling,* which is a play on the `floor()` function.

Suffering through trigonometry

I don't bother to explain trigonometry to you. If your code needs a trig function, you'll know why. But what you probably don't yet know is that trigonometric functions in C — and, indeed, in all programming languages — use radians, not degrees.

What's a radian?

Glad you asked. A *radian* is a measurement of a circle, or, specifically, an arc. It uses the value π (pi) instead of degrees, where π is a handy circle measurement. So instead of a circle having 360 degrees, it has 2π radians. That works out to 6.2831 (which is 2×3.1415) radians in a circle. Figure 11-1 illustrates this concept.

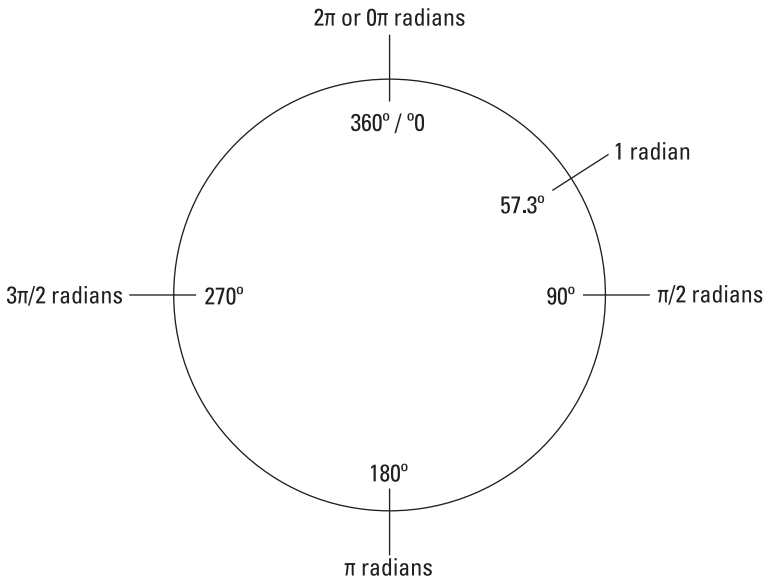

Figure 11-1: Degrees and radians.

For your trigonometric woes, one radian equals 57.2957795 degrees, and one degree equals 0.01745329 radians. So when you do your trig math, you

need to translate between human degrees and C language radians. Consider Listing 11-6.

Listing 11-6: Convert Degrees to Radians

```
#include <stdio.h>

int main()
{
    float degrees,radians;

    printf("Enter an angle in degrees: ");
    scanf("%f",&degrees);
    radians = 0.0174532925*degrees;
    printf("%.2f degrees is %.2f radians.\n",
           degrees,radians);
    return(0);
}
```

Exercise 11-13: Type the source code from Listing 11-6 into your editor. I've split Line 10 so that it's more readable on this page. You don't need to split that line when you type it. Build and run. Test with the value `180`, which should be equal to π radians (3.14).

Exercise 11-14: Write a program that converts from radians to degrees.

Though C has many trigonometric functions, the three basic ones are `sin()`, `cos()`, and `tan()`, which calculate the sine, cosine, and tangent of an angle, respectively. Remember that those angles are measured in radians, not degrees.

Oh, and remember that you need the `math.h` header file to make the compiler happy about using the trig functions.

The best programs that demonstrate trig functions are graphical in nature. That type of code would take pages to reproduce in this book, and even then I'd have to pick a platform (Windows, for example) on which the code would run. Rather than do that, I've concocted Listing 11-7 for your graphical trigonometric enjoyment.

Listing 11-7: Having Fun with Trigonometry

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159
#define WAVELENGTH 70
#define PERIOD .1

int main()
{
    float graph,s,x;

    for(graph=0;graph<PI;graph+=PERIOD)
    {
        s = sin(graph);
        for(x=0;x<s*WAVELENGTH;x++)
            putchar('*');
        putchar('\n');
    }
    return(0);
}
```

Exercise 11-15: Type the source code from Listing 11-7 into your editor. Before you build and run, try to guess what the output could be.

Exercise 11-16: Modify the code from Listing 11-7 so that a cosine wave is displayed. Don't get lazy on me! A cosine wave looks best when you cycle from 0 to 2π. Modify your code so that you get a good, albeit character-based, representation of the curve.

No, Exercise 11-16 isn't easy. You need to compensate for the negative cosine values when drawing the graph.

- One radian equals 57.2957795 degrees.
- One degree equals 0.0174532925 radians.

It's Totally Random

One mathematical function that's relatively easy to grasp is the `rand()` function. It generates random numbers. Though that may seem silly, it's the basis for just about every computer game ever invented. Random numbers are a big deal in programming.

A computer cannot generate truly random numbers. Instead, it produces what are known as *pseudo-random numbers*. That's because conditions inside the computer can be replicated. Therefore, serious mathematicians scoff that any value a computer calls random isn't a truly random number. Can you hear them scoffing? I can.

Spewing random numbers

The `rand()` function is the simplest of C's random-number functions. It requires the `stdlib.h` header file, and it coughs up an `int` value that's supposedly random. Listing 11-8 demonstrates sample code.

Listing 11-8: Now, That's Random

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
int r,a,b;

    puts("100 Random Numbers");
    for(a=0;a<20;a++)
    {
        for(b=0;b<5;b++)
        {
            r=rand();
            printf("%d\t",r);
        }
        putchar('\n');
    }
    return(0);
}
```

Listing 11-8 uses a nested `for` loop to display 100 random values. The `rand()` function in Line 13 generates the values. The `printf()` function in Line 14 displays the values by using the `%d` conversion character, which displays `int` values.

Exercise 11-17: Create a new project by using the source code shown in Listing 11-8. Build and run to behold 100 random values.

Exercise 11-18: Modify the code so that all the values displayed are in the range `0` through `20`.

Here's a hint for Exercise 11-18: Use the modulus assignment operator to limit the range of the random numbers. The format looks like this:

```
r%=n;
```

r is the number returned from the `rand()` function. `%=` is the modulus assignment operator. `n` is the range limit, plus 1. After the preceding statement, values returned are in the range `0` through *n*`-1`. So if you want to generate values between `1` and `100`, you would use this formula:

```
value = (r % 100) + 1;
```

Making the numbers more random

Just to give some credit to the snooty mathematicians who claim that computers generate pseudo-random numbers, run the program you generated from Exercise 11-18. Observe the output. Run the program again. See anything familiar?

The `rand()` function is good at generating a slew of random values, but they're predictable values. To make the output less predictable, you need to *seed* the random-number generator. That's done by using the `srand()` function.

Like the `rand()` function, the `srand()` function requires the `stdlib.h` header, shown at Line 2 in Listing 11-9. The function requires an `unsigned int` value, `seed`, which is declared at Line 6. The `scanf()` function at Line 10 reads in the unsigned value by using the `%u` placeholder. Then the `srand()` function uses the `seed` value in Line 11.

Listing 11-9: Even More Randomness

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    unsigned seed;
    int r,a,b;

    printf("Input a random number seed: ");
    scanf("%u",&seed);
    srand(seed);
    for(a=0;a<20;a++)
    {
```

```
        for(b=0;b<5;b++)
        {
            r=rand();
            printf("%d\t",r);
        }
        putchar('\n');
    }
    return(0);
}
```

The `rand()` function is used at Line 16, although the results are now based on the seed, which is set when the program runs.

Exercise 11-19: Create a new project using the source code shown in Listing 11-9. Build it. Run the program a few times, trying different seed values. The output is different every time.

Alas, the random values that are generated are still predictable when you type the same seed number. In fact, when the value `1` is used as the seed, you see the same "random" values you saw in Exercise 11-17, when you didn't even use `srand()`!

There has to be a better way.

The best way to write a random-number generator is not to ask the user to type a seed, but rather to fetch a seed from elsewhere. In Listing 11-10, the seed value is pulled from the system clock by using the `time()` function.

Listing 11-10: More Truly Random than Ever

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    int r,a,b;

    srand((unsigned)time(NULL));
    for(a=0;a<20;a++)
    {
        for(b=0;b<5;b++)
        {
            r=rand();
            printf("%d\t",r);
        }
        putchar('\n');
    }
    return(0);
}
```

Chapter 21 covers programming time functions in C. Without getting too far ahead, the `time()` function returns information about the current time of day, a value that's constantly changing. The `NULL` argument helps solve some problems that I don't want to get into right now, but suffice it to say that `time()` returns an ever-changing value.

The `(unsigned)` part of the statement ensures that the value returned by the `time()` function is an `unsigned` integer. That's a technique known as *typecasting,* which is covered in Chapter 16.

The bottom line is that the `srand()` function is passed a seed value, courtesy of the `time()` function, and the result is that the `rand()` function generates values that are more random than you'd get otherwise.

Exercise 11-20: Type the source code from Listing 11-10 and build the project. Run it a few times to ensure that the numbers are as random as the computer can get them.

Exercise 11-21: Rewrite your solution to Exercise 8-6 (from Chapter 8) so that a random number is generated to make the guessing game more interesting but perhaps not entirely fair. Display the random number if they fail to guess it.

The Holy Order of Precedence

Before you flee the tyranny of the Unavoidable Math Chapter, you need to know about the order of precedence. It's not a religious order, and it has nothing to do with guessing the future. It's about ensuring that the math equations you code in C represent what you intend.

Getting the order correct

Consider the following puzzle. Can you guess the value of the variable *`answer`*?

```
answer = 5 + 4 * 3;
```

As a human, reading the puzzle from left to right, you'd probably answer `27`: 5 + 4 is 9 times 3 is 27. That's correct. The computer, however, would answer `17`.

The computer isn't wrong — it just assumes that multiplication is more important than addition. Therefore, that part of the equation gets calculated first. To the computer, the actual order of the values and operators is less important than which operators are used. To put it another way, multiplication has *precedence* over addition.

You can remember the basic order of precedence for the basic math operators like this:

First: Multiplication, Division

Second: Addition, Subtraction

The clever mnemonic for the basic order of precedence is, "My Dear Aunt Sally." For more detail on the order of precedence for all C language operators, see Appendix G.

Exercise 11-22: Write a program that evaluates the following equation, displaying the result:

```
20 - 5 * 2 + 42 / 6
```

See whether you can guess the output before the program runs.

Exercise 11-23: Modify the code from Exercise 11-22 so that the program evaluates the equation

```
12 / 3 / 2
```

No, that's not a date. It's 12 divided by 3 divided by 2.

Forcing order with parentheses

The order of precedence can be fooled by using parentheses. As far as the C language is concerned, anything happening within parentheses is evaluated first in any equation. So even when you forget the order of precedence, you can force it by hugging parts of an equation with parentheses.

Math ahead!

Exercise 11-24: Code the following equation so that the result equals 14, not 2:

```
12 - 5 * 2
```

Exercise 11-25: Code the following equation (from Exercise 11-22) so that addition and subtraction take place before multiplication and division. If you do it correctly, the result is 110:

```
20 - 5 * 2 + 42 / 6
```

- In the future, the code you write may deal more with variables than with immediate values, so you must understand the equation and what's being evaluated. For example, if you need to add the number of full-time and part-time employees before you divide by the total payroll, put the first two values in parentheses.
- Beyond the order of precedence, parentheses add a level of readability to the code, especially in long equations. Even when parentheses aren't necessary, consider adding them if the result is more readable code.